Remember: Read King Chapters 5, 6 and 7
Note Dates: Quiz dates are: 30 Jan, 13 Feb, 11 Mar, 1 Apr.
Question: Is your home computer a true 32-bit machine (like falun) or a 16-bit computer? Do you know how to make your C/C++ compiler support 32-bit operations?

C Programming

C Control Structures

* There are a number of control structures in C, we will briefly list them here:

1. **Compound statement**

```
{   statements   }
```

2. **Conditional Statement**

```
if ( expression )
    statement
```

```
if ( expression )
    statement
else
    statement
```

```
if ( expression ) {
    statements
} else {
    statements
}
```

3. **While statement**

```
while ( expression )
    statement
```

```
while ( expression )
{   statements   }
```

4. **Do-while** or Do-until statement

```
do
    statement
while ( expression ) ;
```

The iteration continues when the expression evaluates to a non-zero value, a zero value ends the iterating

5. **For statement**

```
for ( expression1 ; expression2 ; expression3 )
    statement
```

All three expressions are optional
1. **expression1** is evaluated once at the start of the loop
2. **expression2** is evaluated at the start of each iteration, if the value of this expression is 0 the loop is exited
3. **expression3** is executed at the end of each iteration.

The **for** statement is equivalent to:

```
expression1;
while ( expression2 ) {
    statement
    expression3;
}
```

6. **Switch** statement

```
switch ( expression ) {
    case constant-expression : statement
    case constant-expression : statement
        .
        .
        .
    default : statement
} ;
```

The expression in the switch must evaluate to an integer value, all the case labels must also have integer values - when the switch statement is entered the expression is evaluated and control transfers to the matching case (like a goto statement), if there is no matching case default is used, the default case is optional. Control flows from one case to the next, the statement is not automatically exited when the next case is reached

7. **Break**

```
break ;
```

The **break** statement causes the termination of the smallest enclosing **while**, **do**, **for** or **switch** statement A **break** is commonly used to separate the cases in a switch statement

8. **Continue**

continue ;

Causes control to transfer to the loop continuation (end of the loop) portion of the smallest enclosing **while**, **do** or **for** statement This statement is used to transfer control to the next iteration of the loop, that is, terminate the current iteration and start on the next iteration

Example

* In the example we use the following strategy to read from a file or terminal

```
read first line
while ( not end ) {
    process the line
    read next line
}
```

* The main problem with this approach is we need two read statements, an alternative schema (plan) uses the break statement

```
while ( TRUE ) {
    read next line
    if ( EOF )
        break;
    process the line
}
```

Make Facility

* The make facility is one of the most important tools that you will learn in this course. You will use it in all your other computing science courses
* **make** will automatically construct the executable form of a program from a description of the source files and the dependencies between the files
* Once you have produced a Makefile for a program you don't need to remember how to compile it, or even the files you have changed. **make** handles these details automatically for you
* The main input to **make** is a Makefile (or makefile). It describes how to compile and link the program
* The Makefile is divided into two main parts:
  1. macro definitions
  2. dependency declarations

Macro Definitions

* Macros are used extensively in UNIX and C programming, get used to them
* A macro is a text replacement facility
* A macro has a name, which is a character string, and a body, which is also a character string, wherever the name occurs in the file it is replaced by the body of the macro
* In **make** we use macros to describe the compiler we are using, the compiler options, and the list of files we need. It allows us to parameterize the Makefile

* A macro definition has the following format:

  macro-name = macro-body

* A macro definition can appear anywhere in a Makefile
* In **make**, a macro is invoked by mentioning its name, preceded by a $, if the macro is more than one character long the name must be placed in parenthesis
* In our example makefile we have

  FILES = main.o phone.o

* When we use $(FILES) in the rest of the Makefile it is replaced by the two file names, as we add more files to the program we only need to change the line where FILES is declared, the rest of the Makefile stays the same - this saves time and reduces mistakes!!
* Similarly we have the lines

  CC = gcc
  CFLAGS = -Wall -ansi

* The first line specifies the C compiler that we use, and the second line defines the compiler flags. Again we can quickly change the compiler options without scanning through the entire Makefile for occurrences.

Dependency Declarations

* The dependency declarations tell **make** which files depend on which other files
* In the case of our example, phone (the executable for our program) depends on both main.o and phone.o; we indicate this in the following way:

```
phone : main.o phone.o
```

* The file on the left of the **:** depends on the files listed on the right
* Similarly, both main.o and phone.o depend on phone.h, there are two ways that we could specify this

```
main.o :    phone.h

phone.o :   phone.h
```

or

```
main.o phone.o :    phone.h
```

* After each dependency declaration we can list the commands that construct the dependent files, these commands must be indented, there must be either spaces or tabs at the beginning of the line.

WARNING: Old versions of make require tabs.

* So in order to create phone, we use the following specification to link the object modules together.

```
phone : main.o phone.o
        $(CC) $(CFLAGS) -o phone main.o phone.o
```

* When we need to re-create phone, **make** will automatically execute the **gcc** command
* When is a dependency declaration used?
* When make is started it looks at the time of last modification of all the files mentioned in the Makefile. If the last time of modification of a file on the right side of a dependency declaration is more recent than one of the files on the left side then the commands are executed!
* In our example, if either main.o or phone.o is more recent than phone, the gcc command will be executed, otherwise **make** knows that phone is up-to-date and does nothing
* How does **make** know how to make main.o and phone.o, there are no commands associated with their dependencies?
* Make has a set of default rules that know about Unix's file naming conventions, we use suffixes to indicate the type of file, start with a dot (**.**) and have one or more letters, the common ones are:

  name**.c** a c program
  name**.h** a header file
  name**.o** an object file
  name**.s** an assembler file
* Make uses the file suffixes to determine how to make a file, if make needs a **.o** file and can find a **.c** file with the same prefix, it knows that it can use the C compiler to produce the **.o** file

* For example if name.o is required, mentioned on the right side of a dependency rule, but there is no rule for name.o then **make** will look for a file named name.c, and if it finds one will run the C compiler on it, the use of **default rules** reduces the typing you must do *(but I avoid them)*

* If **make** is started without arguments, it will use the first dependency declaration as its target. That is, it will make the file on the left side of the first dependency rule in the Makefile
* In our example Makefile, **phone** is the first file mentioned, so by default make will execute that command first
* You can specify the target when you call make, if we enter the command

make main.o

* then make will only run the c compiler on main.c to produce main.o, it will not produce a new version of phone
* The **most common** file that you produce should be in the **first** dependency declaration, so you don't need to mention it each time you use make
* Besides making the executable of a program there are other standard things that are placed in a Makefile, some of these operations include removing temporary files, installing the executable in a standard place, running standard test cases, and printing the program

* We could add the following rules to our Makefile:

```
clean:  $(FILES)
        rm $(FILES)

install: phone
        cp phone $HOME/bin/phone
```

* When we execute the command

make clean

All the .o files for the phone program will be deleted. Note that a file called clean will not be produced, therefore, each time we execute this command the .o files will be deleted - **make** does not check whether it has actually created the file that was on the left side of the rule!